Idioms for Building Software Frameworks in AspectJ

Stefan Hanenberg[1] and Arno Schmidmeier[2]

[1]Institute for Computer Science
University of Essen, 45117 Essen, Germany
shanenbe@cs.uni-essen.de

[2]AspectSoft,
Lohweg 9, 91217 Herbruck, Germany
A@schmidmeier.org

ABSTRACT

Building applications using AspectJ means to design applications build upon the new language features offered in addition to Java. The usual argumentation that AspectJ permits a better separation of concerns in contrast to the traditional static typed object-oriented code might be valid, but does not prevent developers to misuse these language features. What's needed is a discussions of how to apply the language features of AspectJ to achieve good designed applications. In this paper we propose four idioms whose application turned out to result in good designed application in an appropriate context.

1. INTRODUCTION

AspectJ comes with a number of (more or less) new language features which try to tackle the problem of crosscutting code. Although the impact of these features on the object-oriented code is already analyzed like for example in [6] it is not clear how to apply these features to new problems. However, as noted by R. W. Floyd : "*To persuade me of the merit of your language, you must show me how to construct programs in it. I don't want to discourage the design of new languages; I want to encourage the language designer to become a serious student of the details of the design process*" [2, p. 460].

In this paper we continue to describe idioms which turned out to be used in good designed AspectJ applications and try in this way to encourage the usage of AspectJ in large scale applications. In [4] we already proposed some idioms which were closely related to the language features of AspectJ. Here we propose idioms which seem to be somehow more advanced. The idioms where successfully used in a large scale AspectJ project on Enterprise Application Integration (EAI) systems [9]. In [4] we already discussed the relationship between the proposed idioms and patterns. One of the major points in this discussion was, that the proposed idioms did not have the pattern format. However, in this paper we still neglect to put the idioms in such a format because of two reasons. First, we feel that it is still more important to discuss typical design decisions in aspect-oriented languages than to claim that a number of good patterns are found. And second, it is still not yet clearly determined what language features an aspect-oriented language will provide in the future: the provided language features still evolve from version to version. Hence, a collection of good design decisions might be no longer valid in the future because of language changes in AspectJ.

Furthermore, we do not give an example for each proposed idiom. Instead, we discuss the design of an existing real-world example which was influenced by the here proposed idioms at the end of the paper.

In section 2 to 5 we discuss four different idioms. We concentrate in this discussion on the core ingredients of those idioms, their advantage and their consequences. In section 6 we discuss a real-world example where such idioms were used. Since the example comes from quite a large context we just can give a small glimpse of it and we only concentrate on the considerations which lead to the final design. In section 7 we conclude the paper.

2. TEMPLATE ADVICE[1]

A template advice is used whenever additional behavior which should be executed at a certain join point contains some variabilities. That means the code to be executed consists of a fixed and a variable part whereby the variable part changes from application to application. A template advice corresponds directly to the *template method design pattern* [3] whereby the template is specified inside an advice instead of a method.

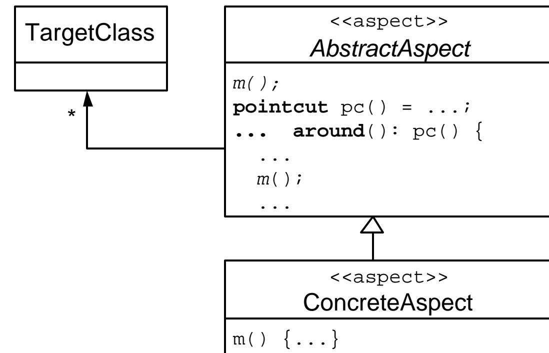

Figure 1. Template Advice

Whenever the problem is given that the code to be executed at certain join points is partly known and fixed, but might contain (depending on the concrete join point) some variabilities, the designer has to decide how to consider such variabilities. In case

[1] It should be noted that the name *template advice* has been already used in [4] for a different idiom. We regard the name of the idiom in [4] a little bit misleading and renamed it to *import pointcut* since this metaphor seems to describe its usage more appropriate.

the join points are well-known, it is possible to implement a number of different aspects for each of those different kinds of join points. The disadvantage of this approach is, that all those different join points usually have some commonalities. This commonalities (which means, that some common pointcut specifications are used) depict redundant code inside the application. The problem can be reduced by using the *composite pointcut idiom* [4]. However, the problem is still, that the advice contains redundant code. This redundant code depicts those part which are constant in each occurrence of the advice. To reduce this redundancy the stable part of the advice is used as a template and the variable part is put into a method. The property of AspectJ that advice cannot be refined in subaspects means in such a situation that the decision which part of the code is fix is final: it is not possible to refine advice incrementally. Instead, the aspect needs to be refactored.

The ingredients of the template advice are (in correspondence to *template method* [3]):

- *Abstract aspect*: the aspect that contains declarations of a number of primitive operations which are defined in sub-aspects. Usually, these methods are abstract, that means they are just declared but not defined. Furthermore, the abstract aspect contains the advice (which is called the template advice) which contains the invocations of the primitive operations. Usually the pointcut referred by the template advice is abstract.
- *Concrete aspect*: The concrete aspect defines or overrides the primitive operations of the abstract aspect and implements in that way the aspect-oriented adaptation of the target classes.

It seems questionable if the template advice is really a specific AspectJ idiom since it is very similar to the *template method*. However, we regards it as an specific idiom, because the consequences of using a template advice are quite different than the usage of a template method. First, in AspectJ only abstract aspects can be extended by further aspects. That means, when the corresponding aspect is written it must be clear whether or not a contained advice inside the aspect tends to be a template advice or not. Using pure object-oriented language features in a language supporting late binding this question does not have to be answered. For example in Java or Smalltalk almost every method can be overridden by a subclass. That means every method inside the superclass which contains invocations of an overridden method depicts a potential template method. That means methods might become template methods because of an incremental modification of the class structure. Hence, the preplanning problem of design patterns as mentioned in [1] is not that significant for a *template method*. On the other hand, because of the limited possibilities of incremental aspect refinement in AspectJ this problem is more present in a template advice. Hence, the consequences of using a template advice are much more restrictive.

The consequences of using a template advice are:

- *Separation of fixed and variable part of crosscutting code*: the advice depicts the fixed part of the crosscutting code, while the abstract method depicts the variable part which can be refined according to the special need.
- *Limited incremental refinement*: since AspectJ does not permit to refine advice directly (via overriding) the advice implementation is usually fixed.
- *Conflict handling*: if there are more than one concrete aspect which refer to at least one common join point the developer need to determine which advice should be executed. This can be either realized by further idioms, or by an explicit usage of dominate relationships between aspects.
- *Limited knowledge on aspects internals required*: the adaptation of the aspect behavior just depends on the concrete method definition. Hence, the developer performing the aspect adaptation only needs little knowledge about the concrete pointcut or the advice internals. However, a detailed description of the contract belonging to the abstract method is needed.
- *Lost access to introspective facilities*: since the reflective facilities of AspectJ are just available inside an advice there is no possibility to refer inside the method to the execution context. This must be considered during the design. In case the execution context might be needed, it has to be passed as a parameter.

Template advice usually occur together with *composite pointcuts* [5] where the concrete aspect defines the component pointcuts. Also, template advice are often used in conjunction with *pointcut methods* and *chained advice* (see section 3 and 4) where in both cases the concrete aspect refines the pointcut definition. Hence, different implementation of template advice usually differ in their handling of the corresponding pointcut.

It should be noted that template advice is a very generic idiom which builds in conjunction with template method and composite pointcut the fundament of aspect-oriented frameworks in AspectJ. It can be compared to [7] whose analysis of software frameworks is based on the distinction between hook and template coming mainly from the template method design pattern.

3. POINTCUT METHOD

A pointcut method is used, whenever a certain advice is needed whose execution depends on runtime specific elements which cannot or only with large effort expressed by the underlying pointcut language.

The pointcut language of AspectJ is quite expressive. Dynamic pointcuts like `args(..)` permit to specify join points which are evaluated during runtime and permit in that way to specify a large variety of crosscuttings. Typical examples where dynamic pointcuts are used are the simulating dynamic dispatching on top of Java (cf. e.g. [9]). However, sometimes the decision of whether or not a corresponding advice should be executed is not that easy to specify inside a pointcut definition. Such a situation is usually

given if the advice execution depends on a more complex computation or includes a invocation history of the participating objects.

The usage of `if` pointcuts can reduce this problem. However, `if` pointcuts are somehow ugly since they permit only to call static members of the aspect. Furthermore, the usage of `if` pointcuts usually reduces the reusability of the enclosing aspect, because they are usually very specific to a small set of join points. Usually, the usage of the pointcut language seems to be inappropriate when the decision whether or not a corresponding advice should be executed can be better expressed by methods than the pointcut language. In these cases the usage of a pointcut method is appropriate.

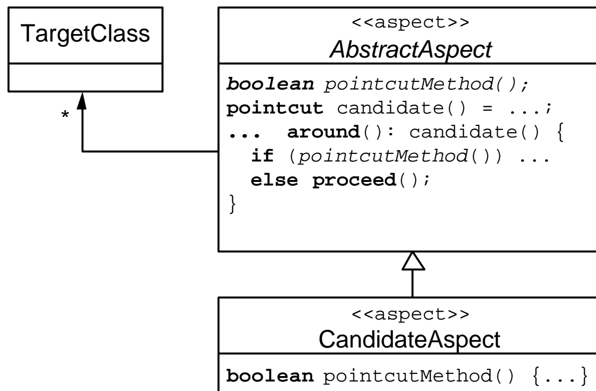

Figure 2. Pointcut Method

The ingredients of a pointcut method are:

- *Candidate pointcut*: the pointcut which determines all potential join points where additional behavior might take place. However, the pointcut definition includes more join points than needed to perform the aspect specific behavior.
- *Pointcut Method*: the method which is invoked from inside the advice to determine whether or not the advice should be executed. Typically the return type of a pointcut method is boolean.
- *Conditional Advice*: the advice which contains the behavior which might be executed at the specified join points. The additional behavior is conditional executed depending on the result of the pointcut method.
- *Candidate Aspect:* the aspect which refines the pointcut method.

Implementations of pointcut methods vary in a number of ways. First, usually a pointcut method's return type is boolean. That means a pointcut method only determines whether or not the additional behavior specified inside an advice should be executed. On the other hand, a pointcut method can also include just any computation whereby the conditional execution of the advice depends on the pointcut methods result (and any other context information). That means the decision whether or not the advice should be executed not only depends on the pointcut method itself.

Another important issue is how the computation of the pointcut method depends on the execution context of the application. Usually context information are directly passed by the advice to the pointcut method. That means the referring pointcut either passes some parameters to the advice or the advice extracts context information using the introspection capabilities of AspectJ like `thisJoinPoint` or `thisStaticJoinPoint`. Another possibility is, that the aspect itself has a state that is set by the application's execution context. The pointcut method can decide because of this state whether the advice should be executed or not.

An advantage of using a pointcut method is its adaptability by aspects: it is possible to specify further advice which refine the pointcut method outside the aspect hierarchy. That means, the condition whether or not an advice should be executed can be modified incrementally. In case the pointcut is hard-coded by using the pointcut language such an extension is not that easy. It assumes a corresponding underlying architecture or *rules of thumb* like discussed in [5].

The consequences of using a pointcut method are:

- *Hidden pointcut definition*: the user which specifies the pointcut method does not need to understand the implementation of the whole pointcut. He just needs an acknowledgement that at least all join points he is interested in are specified by the pointcut.
- *Parameter passing*: to determine whether or not the advice should be executed, the pointcut method needs some inputs. This might be for example property files, or (which is more usual) parameters which are passed from the pointcut to the advice and then from the advice to the pointcut.
- *Possible late pointcut refinement*: the pointcut method can be refined by further aspects.
- *Default advice behavior*: in case the conditional advice is an after or around advice, it is necessary to specify any default behavior. Around advice usually call `proceed`, while after advice usually pass the incoming return value.
- *Little knowledge about advice internals needed*: when specifying the pointcuts it is not necessary to understand all internals of the advice. Usually it is enough to have a description in natural language what kinds of join points can be handled by the advice and what kind of impact the advice has on the join point.

The pointcut method idiom is similar to the *composite pointcut* [5]. Both divide the pointcut into a stable and variable part (usually a composite pointcut it used in conjunction with a inheritance relationship between aspects). The difference between both is, that for adapting a composite pointcut the application of an inheritance relationship between aspects is necessary. This also implies that a composite pointcut has some preplanned variabilities (which are usually component pointcuts). A pointcut method does not directly depend on an inheritance relationship. The refinement might be either achieved via inheritance or by an advice. In the first case a pointcut method plays the role of an

abstract method inside a template advice. In the latter case, a pointcut method is often refined by a chained advice.

4. CHAINED ADVICE

Whenever there is (extrinsic) behavior of objects which is regarded to be somehow fragile what means it seems as if these methods might change because of a number of different decisions and furthermore by a number of different aspects the usage of chained advice is recommended.

Object-orientation already offers to extend the behavior of objects via the inheritance mechanism. Often this extension is based on a *template method* [3] where the pattern's abstract method already contains a concrete implementation. However this does not really solve the adaptation problem: the adaptation is achieved by inheritance and that implies a new class has to be created which overrides and adapts a known one. Furthermore, it must be guaranteed that the request for creating new objects must be redirected to the new class in certain situations. If (for the original classes) no *creational patterns* [3] where used such a task tends to be error-prone and the resulting design is usually unacceptable. In such cases, where an application's behavior at (at least) one join point depends on a number of concerns those concerns are usually not orthogonal, but interact in some way. That means, the new behavior should be modularized in separate aspects, but the relationship between such non-orthogonal concerns must be considered. In such cases we propose to apply the chained advice idiom.

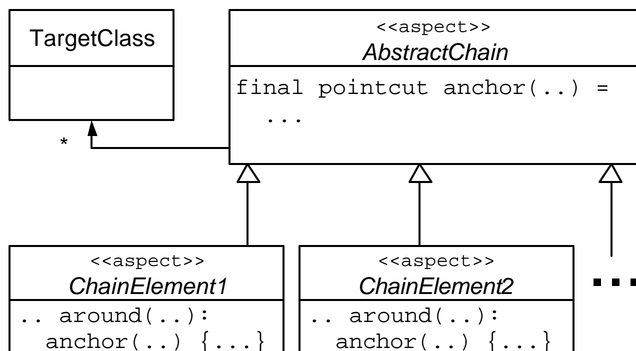

Figure 3. Chained Advice

The ingredients of a chained advice are:

Abstract chain: the aspect containing the anchor pointcut.

Anchor pointcut: the pointcut which is used by every advice within the chain. We call this the anchor pointcut, because each chain of advice is anchored at each join points part of this pointcut definition.

Chain element: The aspects extending the abstract chain and containing the advice which refers to the anchor pointcut. The chained advice have a predefined order. Usually each advice contains a mechanism to redirect the execution to a different advice.

In contrast to the previous mentioned idioms, a chained advice comes with a number of different implementations. On the one hand it is not necessary that the pointcut is inherited from a super-aspect. Instead, we found either the usage of static pointcuts, or even more complex aspect hierarchies than illustrated in figure 3. We found implementations where the chain was realized by an ordinary `proceed`-call, in other cases we found more complex pointcut definitions (that means each chain element offers a join point used by the following chain element). Also, in many cases the execution of chained advice is mutually exclusive, than means at most one chained advice is executed. But there are situation where more than one chain element is executed. What kinds of chained advice should be used depends on the concrete situation.

The way how the mutually exclusive advice were realized differ in different applications. On the hand (as we will illustrate in the final example) *pointcut methods* were used, in other cases ordinary advice in combination with *composite pointcuts* [5] were used. Both implementations have their pro and cons. The advantage of the first approach is that aspects do not need to have any knowledge about each other, i.e. their implementations do not depend on each other. But this also means that the advice execution order has to be controlled in some way. The latter approach assumes an explicit dependency of each advice.

The consequences of using the chained advice idiom are:

- *Separate concerns for each advice*: each advice represents certain behavior coming from different concerns within its own module.
- *Independent composability*: certain elements within the chain can be composed independent of each other. The level of independence of each chain elements depends on the underlying implementation. The major benefit is usually, that new chain elements can be added without the need to perform any destructive modifications within existing chain elements.
- *Parameter passing*: a mechanisms is needed to pass the responsibility from one chain element to another.
- *Default behavior needed*: often chained advice need to provide a default behavior at the anchor join points.

Chained advice make often use of *pointcut methods* to determine whether or not a chain element should be executed. Furthermore, chained advice often make use of *composite pointcuts* to reduce redundant pointcut definitions.

5. FACTORY ADVICE

Whenever the object creation of certain object depends on specific aspects which might vary from application to application or the execution context of an application, the usage of a *factory advice* is recommended.

A factory advice is an advice which is responsible for the object creation. It looks similar to the well-known design pattern *factory method* [3]. The argumentation why we still regard this a specific idiom in AspectJ is similar to the argumentation in section 2: the consequences of using a factory advice differ widely from the factory method.

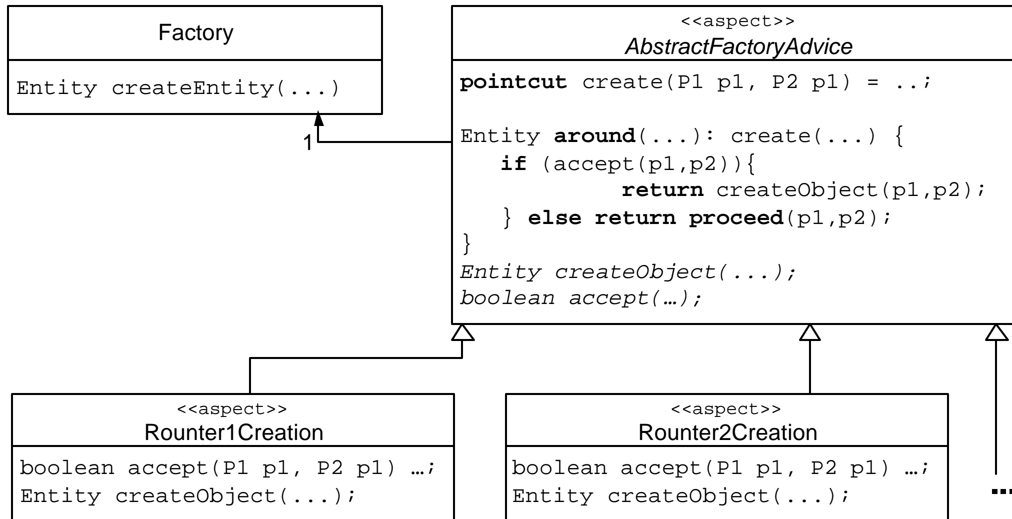

Figure 5. Example: Object creation in large scale frameworks

Whenever the creation of objects depends on certain aspects (and there might be more than one aspect) and such object creation might differ in different applications or different execution contexts it is usually not appropriate only to intercept the object creation using a pointcut to the constructor and then redirecting the creation using an around advice. The problem in such a context is usually the restriction that around advice need to return the same type than its join points.

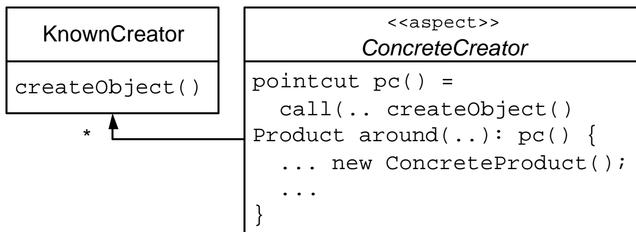

Figure 4. Factory Advice

The ingredients of a factory advice are:

- *default create method*: the method which is invoked by a client to request a new object. Usually, the method just return a null object.
- *a concrete creator*: the aspect which contains a pointcut to the default create method and the specification what product should be created.
- *abstract product*: the product expected by the client. Usually the factory advice redirects the creation of the abstract product to a different class extending the abstract product.

The relationship between a factory advice and the usual application of advice can be seen like the relationship between the *factory method design pattern* [3] and the *template method* [3]: although both are similar in their relationship of hook and template their differ mainly in the way their intention.

The consequences of using a factory advice are:

- *deferred object instantiation*: the aspect instantiation is no longer hard coded inside the object structure, but moved to the aspect definition. That means the instantiating aspect must be woven to the application to guarantee its correctness.
- *specified default behavior*: an advice factory assumes a specification of a default behavior of the default create method. Usually, the advice overrides the whole behavior specified there. But there are situations where the default create method contains some meaningful code and the aspect code is just executed in "special situations".
- *composability*: The advice factory permits to exchange the object creation process without performing destructive modifications within the object structure.

Factory are often used as chained advice in cases where the object to be instantiated depends on some execution context. In this way the factory advice looks even more like the abstract factory design pattern [4].

6. Example

Object oriented component frameworks suffer always from the problem of the construction of new component instances. The creational patterns in [4] reduce but do not really solve the problem. Each combination of these patterns violates at least in one point the principle of "need to know", which leads to somehow non transparent dependencies. Each component can know everything from the framework but not the other way round. When the framework is responsible to constructing new component instances, the framework needs to know the component. Delegating this kind of knowledge to framework configuration files doesn't solve that problem either. This approach contains several other drawbacks: it is impossible to implement the component in plain Java, a combination of Java and XML is needed, several checks which modern compiler can perform during compile time are no longer possible, code patterns

which enforce all configurations are not possible, this approach is not valid for high performance applications, because of the additional overhead caused by the required use of the Java Reflection API.

It is desirable, that every component connects itself to the construction mechanism. We present a solution of this problem as an example of a combination of the discussed idioms, which has been applied in the EOS-product family [10].

The core functionality is that dependent of the passed parameters the component decides on its own if it should be instantiated or not. That means, it depends on the framework configuration what objects have to be instantiated an in such a situation the application of an advice factory is appropriate. That means the request of an object creation is passed to a certain *default creation method* (we neglect here the implementation of corresponding pointcut create). However, the decision of what concrete product should be created depends on the one hand on the passed parameters and on the other hand on the available components inside the framework.

Since it is possible to specify all join points and it depends on the installed components whether or not they should be instantiated we decided use a *poincut method* inside the *advice factory* as illustrated in figure 5. Clients request a new abstract product (of type `Entity`) from the factory (which is in the concrete example an object). The factory object's default create method contains a dummy implementation. The concrete creator defines a pointcut for this method and defines a template advice and a pointcut method. The pointcut method `accept` specifies whether or not a concrete aspect should be responsible for the object creation or not. The abstract method `createObject` is overridden by concrete aspects and creates a concrete product.

In the here mentioned context we realized the concrete aspects as *chained advice* where each installed component comes with its own chain element for object creation. The reason for it is, that the fixed part of the template advice can be easily implemented as a chained advice and the responsibility which chain element creates the object lies in each chain element's `accept` method. Since the template advice either invokes `createObject` or proceeds with the join points execution all chain elements are mutually exclusive. Under the assumption that each element's pointcut method `accept` is adequately implemented there is no need determine any domination of the aspects.

Since the creational process differed widely from entity to entity we decided the implement the object creation as the abstract method inside the template method idiom.

7. CONCLUSION

In this paper we demonstrated a small collection of idioms we found frequently inside AspectJ applications and demonstrated an example which illustrated the usage of the idioms. The intention of the paper is to demonstrate "good design decisions" in AspectJ and discuss their advantages and disadvantages.

Although we found implementations of the here described idioms in current AspectJ projects we are aware of the fact, that there is no such clear distinction between the here described idioms and the known GoF design patterns *template method*, *chain of responsibility*, *abstract factory* and *factory method*. In that way it looks like the here proposed idioms are more a implementation of known design patterns as e.g. proposed in [6]. On the other hand the consequences of each of the idioms is quite different from the consequences of using the GoF patterns. Such consequences are mainly determined by the restriction that concrete aspects cannot be extended and advice cannot be overridden.

Nevertheless, the *pointcut method* seems to be an idiom which is highly related to an aspect-oriented language features and seems in that way rather a "pure aspect-oriented idiom" than the others. However, we think that the here described idioms are good examples of good AspectJ design which were successfully used and should be therefore considered when designing AspectJ applications if the application's context matches the idioms contexts.

8. REFERENCES

[1] Czarnecki, K.; Eisenecker, U. W.: *Generative Programming: Methods Tools and Applications*, Addison-Wesley, 2000

[2] Floyd, R. W.; *The Paradigms of Programming*, Communications of the ACM, Volume 22, No. 8 (1979), pp. 455 – 460.

[3] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[4] Hanenberg, S.; Costanza, P.: *Connecting Aspects in AspectJ: Strategies vs. Patterns*, First Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD'01, Enschede, April, 2002

[5] Hanenberg, S., Unland, R.: *Using and Reusing Aspects in AspectJ*. Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA, 2001

[6] Hannemann, J., Kiczales, G., *Design Pattern Implementations in Java and AspectJ*, OOPSLA 2002.

[7] Pree, W.: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, Reading, 1995.

[8] Schmidmeier, A.; Hanenberg, S.; Unland, R.: *Implementing Known Concepts in AspectJ*, 3rd Workshop on Aspect-Oriented Software Development of the German Informatics Association, March, 2003

[9] Sirius GmbH, Enterprise Object System, *EOS, Functional Product Overvie*w, EOS Core System Version 3.5, 2002